# 第9讲 扫描程序设计

**教学与实践目的：**学会主机及端口等一些基本的网络扫描程序设计技术。

目标主机扫描是网络功防的基础和前提，扫描探测一台目标主机包括：确定该目标主机是否活动、目标主机的操作系统、正在使用哪些端口、对外提供了哪些服务、相关服务的软件版本等等，对这些内容的探测就是为了“对症下药”，为攻防提供参考信息。

对主机的探测工具非常多，比如大名鼎鼎的nmap、netcat、superscan，以及国内的x-scanner等等。我们自己动手做简单扫描软件或工具，用于加深对网络编程的理解。

**知识点**：进一步理解 new Socket()、Process类和ICMP。

## 一、程序设计第一步：主机扫描（远程主机探测）

（1）通过指定的IP地址范围，发现该范围中活跃的主机，例如指定192.168.0.15-192.168.1.100范围。

新建Java包，命名chapter09，创建主机扫描程序HostScannerFX.java，窗口界面如图9.1所示，并在“主机扫描”按钮中设置主机探测关键代码，例如类似代码：

InetAddress addr = InetAddress.getByName(host);//host为IP地址

boolean status=addr.isReachable(timeOut);// timeOut为等待时间(毫秒)

若status的值为真，则表示该主机是活跃的，否则可能不存在或离线。

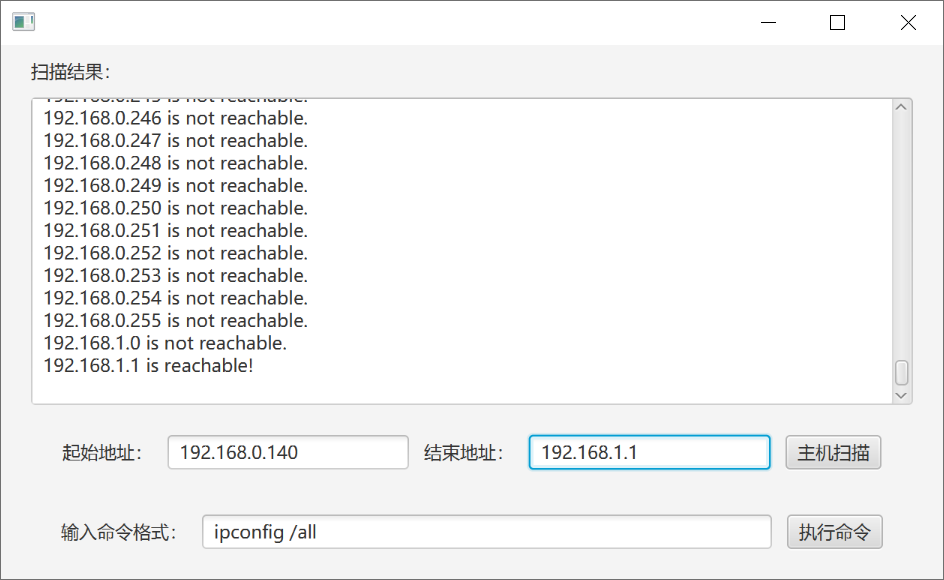


图9.1 主机扫描窗体

在主机扫描的过程中就要涉及到指定ip地址范围的循环遍历，大家思考如何实现。最容易想到的思路可能就是将起始范围的两个ip地址分别拆分成字符串数组，再进行相关的匹配、多重循环等操作。这个过程实际是繁琐低效的。

更好的一种思路就是将ip地址转换为整数形式，一个网段就是0-255之间，刚好就是1个字节的范围，四个网段表示就需要四个字节，刚好是一个int整数的范围。在两个整数之间循环遍历，循环体中将每一个整数转换回ip地址进行处理即可。所以关键就在于实现ip与整数之间的互相转换，基本原理阐述如下：

假设ip地址为 192.168.234.3：

每个网段都用二进制表示： 192(10) = 11000000(2) ; 168(10) = 10101000(2) ; 234(10) = 11101010 (2) ; 3(10) = 00000011(2) 。

所以连在一起就是：11000000101010001110101000000011，对应的int数字就是-1062671869。

具体算法分析：

192左移24位： 11000000 00000000 00000000 00000000

168左移16位： 00000000 10101000 00000000 00000000

234左移08位： 00000000 00000000 11101010 00000000

003左移00位： 00000000 00000000 00000000 00000011

将之拼装即按位或结果：11000000 10101000 11101010 00000011

即 -1062671869

将int类型的数字转换成ip地址，其实就是上述过程的逆过程（在具体运算中要用到和0xff与的操作）。

这些操作需要用到java的位运算操作，例如左移 << 、右移 >> 、 位与& 、位或 |。

这里给出ip转int的方法ipToInt，大家自行实现对应的int转ip的方法intToIp：

*/\*\*  
 \* 字符串格式ip转整数   
 \*/*public static int ipToInt(String ip) {  
 String[] ipArray = ip.split("\\.");  
 int num = 0;  
 for (int i=0;i<ipArray.length;i++) {  
 int valueOfSection = Integer.*parseInt*(ipArray[i]);  
 num = (valueOfSection << 8 \* (3 - i)) | num;  
 }  
 return num;  
}

**注意**：主机扫描是个耗时操作，为了避免程序失去响应，“主机扫描”按钮中的代码应该放在一个新线程中执行。

（2）可以使用包括PING、TRACERT等命令来实现ICMP相关扫描等功能，当然还有其他有关的命令行程序，可以在我们的程序中实现执行其他外部程序的功能，这样就可以把一些功能集成到本程序。在Java中执行其它应用程序，要用到Process类和Runtime类，大家可以自行查找其相关用法，部分核心代码如下：

try {  
 String cmd = tfCmd.getText();  
 Process process = Runtime.*getRuntime*().exec(cmd);  
 InputStream in = process.getInputStream();

*//由于简体中文系统中，命令行程序输出的中文为gb2312编码，所以这里通过流接管其输出，也需要为gb2312（或gbk）编码*  
 BufferedReader br = new BufferedReader(new InputStreamReader(in, "gbk"));  
 String msg;  
 while ((msg = br.readLine()) != null) {

......

......

String msgTemp = msg;  
 Platform.*runLater*(()->{  
 taDisplay.appendText(msgTemp + "\n");  
 });  
 }  
} catch (IOException e) {  
 System.*err*.println(e.getMessage());  
}

同样，很多命令行程序执行时间较长，例如netstat命令，所以“命令执行”按钮中的代码也需要放在新线程中执行。

## 二、程序设计第二步：端口扫描

扫描目的主机开放的端口是常见操作：攻（寻找目的主机开放的端口）与防（检测本机异常开放的端口）。基本的端口扫描可使用的技术：

i. New Socket(host, port);

ii. TCP Connect端口扫描；

创建端口扫描程序PortScannerFX.java，主界面如图9.2所示。

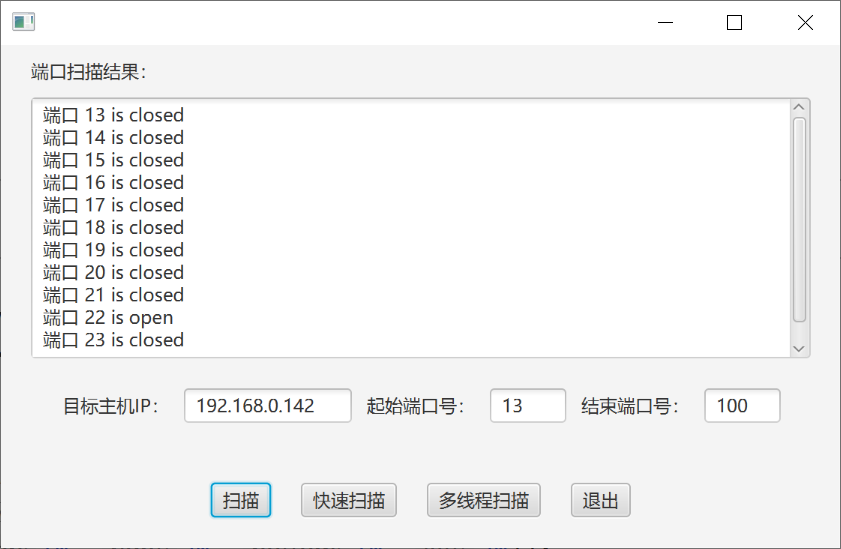


图9.2 端口扫描程序主界面

1. 在“**扫描**”按钮的点击事件响应中添加如下类似代码：

......

try {

Socket socket = new Socket(host,port);  
 socket.close();  
 String msg = "端口 " + port + " is open\n";

*//显示区显示相应信息*  
 ......

} catch (IOException e) {

String msg = "端口 " + port + " is closed\n";  
 ......  
}

用try catch进行捕获，如果能打开，就不会抛出异常，否则利用IOException判定该主机的port端口并没有开放，该方法对于未开放的端口，默认等待时间秒数量级。

（2）上述方法在遇到端口关闭时等待时间过长，时间成本过高。在“**快速扫描**”按钮的事件响应程序中添加如下类似代码：

......

try {

*//空套接字,不尝试连接工作*  
 Socket socket = new Socket();  
 socket.connect(new InetSocketAddress(host,port), 200);  
 socket.close();  
 String msg = "端口 " + port + " is open\n";  
 ......

} catch (IOException e) {  
 ......

}

在这里设定连接等待时间，毫秒时间单位，如设定为200ms。 根据设定的时间尝试连接对方。

（3）**多线程扫描：**即便使用快速扫描的方式，如果扫描整个端口范围，还是耗时较久。可以结合快速扫描，开启多线程加快速度，每一个线程负责扫描一段范围，分而治之加快速度。

在多线程扫描中，出来的结果不是按顺序排列，我们可以考虑没必要把关闭的端口显示出来，只需要显示开放的端口，毕竟我们只关心哪些端口开放了，这时就需要有一个变量来统计已经扫描了多少端口，这样可以通知用户端口扫描结束。这个变量需要考虑并发访问的问题，除了可以使用synchronize方式，还可以使用效率更高的原子操作数据类型，保证线程安全，例如可以定义一个静态的整型原子数据类型变量：

static AtomicInteger *portCount* = new AtomicInteger(0);*//portCount用于统计已扫描的端口数量*

该数据类型提供set(int value)、get()、incrementAndGet()等方法，可以满足我们的要求。记住每次重新扫描的时候要把该变量值重置为0。

下面是实现了Runnable接口的内部类ScanHandler，部分参考代码如下：

*/\*\*  
 \* 用于指定端口范围扫描的线程  
 \*/*class ScanHandler implements Runnable {  
 private int totalThreadNum;*//用于端口扫描的总共线程数，默认为10* private int threadNo;*//线程号，表示第几个线程* public ScanHandler(int threadNo) {  
 this.totalThreadNum = 10;  
 this.threadNo = threadNo;  
 }  
  
 public ScanHandler(int threadNo,int totalThreadNum) {  
 this.totalThreadNum = totalThreadNum;  
 this.threadNo = threadNo;  
 }  
  
 @Override  
 public void run() {

*//startPort和endPort为成员变量，表示需要扫描的起止端口*  
 for (int port = startPort + threadNo; port <= endPort; port = port + totalThreadNum) {  
 try {  
 Socket socket = new Socket();  
 socket.connect(new InetSocketAddress(host,port), 200);  
 socket.close();  
 String msg = "端口 " + port + " is open\n";  
 Platform.*runLater*(()->{  
 taDisplay.appendText(msg);  
 });  
 } catch (IOException e) {  
 */\**

*不输出关闭的端口信息*

*String msg = "端口 " + port + " is closed\n";  
 Platform.runLater(()->{  
 taDisplay.appendText(msg);  
 });\*/* }  
 *portCount*.incrementAndGet();*//扫描的端口数+1* }  
  
 if (*portCount*.get() == (endPort - startPort + 1)) {  
 *portCount*.incrementAndGet();*//加1，使得不再输出下面的线程扫描结束的信息* Platform.*runLater*(()->{  
 taDisplay.appendText("\n----------------多线程扫描结束--------------------\n");  
 });  
 }  
 }  
}

根据上面的内部类的实现，在“**多线程扫描**”按钮中，通过合适的循环语句，生成多个线程并行扫描，请自行完成。

## 三、扫描实验（课堂计分）

主机探测：探测XXX.XXX.XXX.\* ：6060的主机，完成其中交待的任务。具体的IP地址范围课堂上根据具体情况再通知。

提示：扫描过程中要先判定主机可达，再判定该端口开启，然后再使用第3讲的多线程客户端对其发起TCP连接，根据提示完成任务。

## 四、扩展练习：完善主机扫描和端口扫描程序

（1）使用线程技术，使得按钮执行不会卡死主界面，但如果用户再次点击按钮执行新任务，前面的没有完成的线程任务输出就会和后面任务的输出内容混杂显示在一起，请改进这两个程序，使得每次点击按钮时，先关闭之前执行的扫描线程。注意，不要使用Thread.stop()方法，该方法不安全，已经被废弃。大家搜索有关ThreadGroup类、interrupt()、isInterrupted()等用法，实现以上需求；

（2）两个程序各自增加一个“停止”按钮，使得扫描或长时间执行的命令可以停止（实质也是需要关闭对应线程，能完成上面的任务，这个任务也就很容易实现）。

（3）扫描时关闭的端口其实没必要显示，这不是我们关心的结果。但不显示的话，不知道扫描的进度情况，请增加一个进度条（ProgressBar类）功能，动态显示扫描的进度。

ProgressBar可以通过setProgress(double value)来设置进度条的动态变化，value用0到1之间范围表示进度的百分比，getProgress()返回double类型的进度值。效果类似图9.3。

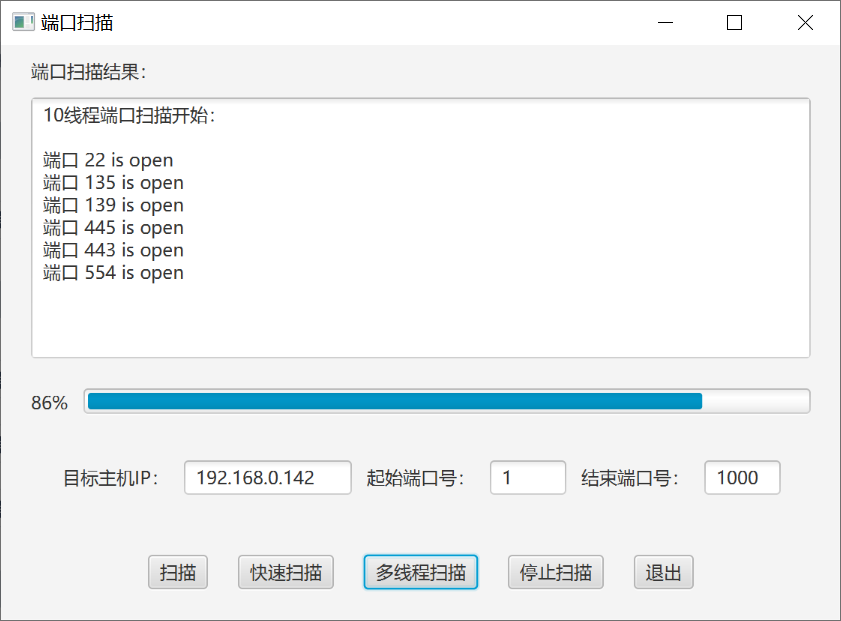


图9.3 带进度条的扫描窗口

（4）主机扫描也可以类似端口扫描，使用多线程方式尝试实现，并同样实现进度条功能（将IP地址用整数的方式表示，就可以类似端口扫描方式：多线程分段扫描。但如果使用ip转int方式，在A网段的127到128之间会出现正负数的跳变，建议将ipToInt方法重构为ipToLong方法）。